1268 UNPL

Bell Laboratories

subject: Variable Length Argument Lists in C

date: June 12, 1978

from: Andrew Koenig
MH 8234
2C-258 x5570
MF: 78-8234-64

MEMORANDUM FOR FILE

0. Introduction

A major problem with the C language is that *printf* cannot be written in C. This is because different invocations of *printf* may appear with different numbers of arguments, and the same argument position may be occupied by arguments of different types from one invocation to another.

However, the definition of *printf* requires that the first argument be a character string, and from this first argument is it possible to derive the number and types of the other arguments (assuming, of course, that the call to *printf* is correctly coded). Furthermore, to allow *printf* to be implemented at all, any C implementation must make information available in some form that will allow *printf* to get at the values of its arguments, though it may be necessary to use a machine language subroutine to do so.

This paper describes an interface to variable length argument lists with the following characteristics:

- It can be implemented on a variety of machines.
- The first argument to a function can be accessed knowing only its type.
- Once argument n has been successfully accessed, argument $n+1$ can be accessed knowing only its type.
- The arguments of a function f can be accessed from within f, or from within a function g invoked directly or indirectly from f.
- In C programs compiled using the PDP-11 UNIX[1] systems, the time required to access an argument using this interface compares favorably with the time required to access an argument without it.

On UNIX systems running on PDP-11 computers, this interface is implemented as a collection of macros.

1. Concepts

For each C implementation, there is some set of information that is necessary and sufficient to access a variable argument list, provided that the type of arguments 1 through n are known before accessing argument $n+1$. This information will be stored in an object of type **va_list**. Thus, after declaring a **va_list** named *ap*, it will be possible to determine the value of the first argument given only *ap* and the type of the first argument.

A **va_list** will also contain a pointer into the argument list; accessing argument n will result in advancing the pointer so that the next access will be to argument $n+1$.

1. UNIX is a Trademark of Bell Laboratories.

The important thing to note about **va_llsts** is that they contain *all* the information necessary to access *all* the arguments. Thus, a function *f* can create a **va_list** for its arguments and pass it to another function *g*, which can then step through the arguments of *f*. An example of the use of this facility is the *printf* family of functions. This family consists of three members: *printf*, *fprintf*, and *sprintf*. In most of these implementations, each of the three functions calls a common sub-function, and it is important for this sub-function to be able to scan the arguments of the parent function.

2. Behavior

Every file that contains functions which are to access a variable argument list must contain the following line:

```
#Include <varargs.h>
```

This line includes macro definitions for the items whose descriptions follow, as well as a **typedef** declaration for va_list.

2.1 va_alist, va_dcl

Va_alist and *va_dcl* are used to form the initial line of the function: the one bearing its name, the type of its result, and the types and names of its arguments. *Va_alist* is a macro which expands into the argument list that the particular implementation will require to allow the function to handle a varying number of arguments, and *va_dcl* is a macro which expands into the declarations appropriate to the argument list, *including* a terminating semicolon if necessary.

Thus, a version of *fscanf* might begin as follows:

```
#include <varargs.h>
int fscanf (va_alist) va_dcl
{
```

2.2 va_list, va_start, and va_end

Va_list is a data type which is defined in *varargs.h*. As mentioned above, it contains all the information necessary to make a single sequential pass through the argument list. A **va_list** is initialized by passing its name to the macro *va_start*. This initialization must be performed in the function whose argument list is to be scanned.

Once the program is done with the argument list, it must call *va_end* with the **va_list** name as argument, to indicate that further use of the va_list is not required.

Thus, our definition of *fscanf* has grown:

```
#Include <varargs.h>
int fscanf (va_alist) va_dcl
{
        va_llst ap;
        va_start (ap);

        The body of the program appears here.

        va_end (ap);
}
```

2.3 va_arg

Va_arg is used to access an argument. Its two arguments are the name of a **va_list** and the data type of the argument which it is desired to access. Recall that a **va_list** contains an indication of where it is in the argument list being scanned; *va_arg* has the side effect of causing the **va_list** to address the next argument in the list.

The most important characteristic of *va_arg* is that it need not appear in the same function as the call to *va_start* that established the **va_list** in question. Since the **va_list** contains all the information necessary to access the argument list, there is no reason why the argument list cannot be accessed in a

sub-function.

Thus, our final version of *fscanf* looks like this:

```
#include <stdio.h>
#include <varargs.h>

int fscanf (va_alist) va_dcl
{
        va_list ap;
        FILE *fp;
        int _doscan();
        int n;

        va_start (ap);
        fp = va_arg (ap, FILE *);

        n = _doscan (fp, &ap);

        va_end (ap);
        return n;
}
```

The two points worth noting about this program are:

1. The instance of *va_dcl* is *not* followed by a semicolon; the *va_dcl* macro will generate one if it is appropriate.
2. The second parameter to *_doscan* is the address of *ap* rather than *ap* itself; this is not a requirement of the method, but may be useful for those implementations in which a **va_list** is, say, a structure containing many elements.

A definition of *_doscan* appropriate for this definition of *fscanf* might be:

```
#include <stdio.h>
#include <varargs.h>

int _doscan (f, a)
{
        FILE *f;
        va_list a;
        char *format;

        format = va_arg (a, char *);

        ...

}
```

3. PDP-11 UNIX Implementation

The PDP-11 UNIX implementation of this facility is all macros, except for a **typedef** declaration for **va_list**:

```
typedef char *va_list;
#define va_dcl int va_alist;
#define va_start(list) list = (char *) &va_alist
#define va_end(list)
#define va_arg(list,mode) ((mode *) (list += sizeof(mode)))[-1]
```

Note first that va_alist is not even a macro; the combined effect of the definition of *va_dcl* and the non-definition of *va_alist* is to cause a function which is to accept a variable argument list to appear as if

it had a single character argument named *va_alist*.

Since PDP-11 UNIX C stores its arguments contiguously on the stack, the address of the current argument is all the information needed to step through the arguments, so **va_list** is simply a character pointer. *Va_start* sets its argument to the address of the (first) argument (with a cast to avoid a complaint from *lint*), and *va_end* does nothing at all.

The most complicated macro is *va_arg*. It must return the value of appropriate mode pointed to by its *va_alist* argument, and increment that argument by the length of that mode. Since the result of a cast is never an lvalue, this is accomplished by determining the amount by which to increment (with **sizeof**) and adding it directly to the character pointer. The resulting pointer is cast into the required mode, and, since it now points one increment too far, a subscript of -1 is used to access the correct value.

4. Notes

4.1 Restricted modes

A trap into which users of this facility are likely to fall is trying to specify a second argument of **char**, **short**, or **float** to *va_arg*. This will not work because arguments are never **char**, **short**, or **float**. Instead, **char** and **short** arguments are converted to **int**, and **float** arguments are converted to **double**. In the UNIX implementation, incorrect specifications of this sort will cause considerable trouble.

4.2 Alignment Problems

The Interdata represents a class of machines on which it will be hard to implement this argument passing technique. The trouble is that although arguments are stored on the stack (as opposed to pointers to arguments being stored), each argument must be aligned on a boundary appropriate to its type. Thus, to advance a pointer to the next argument, one must not only know the current argument's length but also the next argument's alignment requirements.

This implies that an Interdata implementation of *va_arg* would first align the pointer appropriately to the type of the argument being accessed, fetch the argument, and then advance the pointer.

The trouble is that there is currently no way of obtaining the alignment requirements of a data item without knowing the particular implementation. Specifically, there is no way to design a macro which, given a data type, will be able to derive the alignment requirements of that data type. Thus, to make this scheme work properly on the Interdata may require introduction of a companion to **sizeof**, which would yield the alignment stringency of its argument.

Even with this limitation, this method is sufficient to handle Interdata calls in which no argument is a structure; thus it is sufficient for *printf* and *scanf*.

Andrew Koenig

MH-8234-ARK-unix

Copy to
C Standards Task Force
G. L. Baldwin
T. A. Dolotta
F. S. Dworak
R. C. Haight
S. C. Johnson
G. W. R. Luderer
M. D. McIlroy
D. M. Ritchie
B. A. Tague
K. Thompson